Clean for Haskell98 Programmers

– A Quick Reference Guide –

Peter Achten, Radboud University Nijmegen, The Netherlands, P.Achten@cs.ru.nl

July 13, 2007

This note is meant to give people who are familiar with the functional programming language Haskell98 a concise overview of Clean language elements. The goal is to support the reader when reading Clean code. In the table on the other side of this page frequently occurring Clean language elements are summarized, and their Haskell98 counterpart is given next to it.

Obviously, this summary is not exhaustive. Notable Clean language elements that also occur frequently in Clean programs, but that do not appear in this summary are:

Strictness By default, Clean evaluates expressions lazily. Types can be annotated with strictness attributes (`!`) to indicate that they occur within a strict context. This is similar to Haskell98. Within a function definition, `#!` can be used to enforce evaluation of expressions. Haskell98 programmers will likely use the `seq` function.

The uniqueness type system Briefly, types and type variables can be annotated with a uniqueness attribute. This attribute can be `*`, which indicates that the type must occur within a unique context, and `u:`, which is a uniqueness attribute variable, which can be instantiated with `*` or not. The uniqueness type system allows Clean to use the *world-as-value* paradigm for side-effective programming: an interactive program is of type `*World -> *World`, where `World` represent the external environment of the program of which there can be only one, hence its uniqueness attribute. In Haskell98, such a program would have type `IO ()`. The uniqueness type system also allows the Clean compiler to generate efficient code because uniquely attributed data structures can be destructively updated.

Generic programming With generic programming, the programmer defines a small set of instances of a generic function scheme that are used by the Clean compiler to derive for arbitrary data types the corresponding instance. Generic programming in Clean resembles generic programming in Generic H∀skell.

Dynamic types Dynamic types allow the programmer to serialize and de-serialize arbitrary expressions (including functions), resulting in a new value of type `Dynamic`. Dynamic values can be stored on disk.

I hope you enjoy this note and that it will aid you in reading Clean programs.

Clean for Haskell98 Programmers

– Quick Reference Guide –

Clean	Haskell98
Basic types	

`(True,False) :: (Bool,Bool)`
`42 :: Int`
`3.1415926 :: Real`
`'A' :: Char`
`"Hello" :: String`
	`(True,False) :: (Bool,Bool)`
	`42 :: Int`
	`3.1415926 :: Float -- or Double`
	`'A' :: Char`
	`"Hello" :: String`

Type definitions	

`:: T a1...an :== t`
`:: T a1...an = C1 t1 | ... | Cn tn`
`:: T a1...an = { f1 :: t1, ..., fn :: tn }`
	`type T a1...an = t`
	`data T a1...an = C1 t1 | ... | Cn tn`
	`data T a1...an = T { f1 :: t1, ..., fn :: tn }`

Abstract data type definitions	
`definition module M` `:: T a1...an // no implementation`	`module M(T)`
Function types	
`f :: t1 ... tn-1 -> tn | C1 a1 & ...& Cm am`	`f :: (C1 a1, ..., Cm am) => t1 -> ...-> tn-1 -> tn`
Type classes	

`class f a :: t`
`class C a | C1, ..., Cm a`
`instance C t | C1, ..., Cm a where ...`
	`class f a where f :: t`
	`class (C1 a,..., Cm a) => C a`
	`instance (C1 a,...,Cm a) => C t where ...`

as-patterns	
`x=:p`	`x@p`
λ-expressions	
`\p1...pn -> e or: \p1...pn . e or: \p1...pn = e`	`\p1...pn -> e`
Distinction of cases	

`if c t e`
`case e of`
`  p1 -> e1 or:`
`  ...`
`f p1...pn`
`  | c = t`
`  = e`
	`if c then t else e`
	`case e of`
	`  p1 -> e1`
	`  ...`
	`f p1...pn`
	`  | c = t`
	`  | otherwise = e`

List expressions	

`[1:[2:[3]]] :: [Int]`
`[e \\ p1 <- g1 ]`
`[e \\ p1 <- g1 | p ]`
`[e \\ p1 <- g1 , p2 <- g2 ]`
`[e \\ p1 <- g1 & p2 <- g2 ]`
	`1:(2:(3:[])) :: [Int]`
	`[e | p1 <- g1 ]`
	`[e | p1 <- g1, p ]`
	`[e | p1 <- g1, p2 <- g2 ]`
	`[e | (p1,p2) <- zipWith (,) g1 g2]`

Record expressions	

`:: R = { f :: t }`
`r = { f = e }`
`r.f`
`r!f`
`{ r & f = e }`
	`data R = R { f :: t }`
	`r = R {e}`
	`f r`
	`(\v -> (f v,v)) r`
	`r { f = e }`

Record patterns	

`:: R1 = { f1 :: R2 }`
`:: R2 = { f2 :: t }`
`g1 { f1 } = e f1`
`g2 { f1 = { f2} } = e f2`
	`data R1 = R1 { f1 :: R2 }`
	`data R2 = R2 { f2 :: t }`
	`g1 (R1 {f1=x}) = e x`
	`g2 (R1 {f1=R2 {f2=x}}) = e x`

Array expressions	

`:: A :== {t}`
`a = {v0,...,vn-1}`
`a = {e \\ p <-: a} (see list comprehensions)`
 	
`a.[i]`
`a![i]`
`{a & [i]=e}`
	`type A = Array Int t`
	`a = array (0,n-1) [(0,v0),...,(n-1,vn-1)]`
	`a = array (0,length a-1)`
	`  [e | (i,a) <- zipWith (,) [0..length a-1] a]`
	`a!i`
	`(\v -> (v!i,v)) a`
	`a//[(i,e)]`

Comments	
`// single line comment` `/* multi-line, /* nested, */ comment */`	`-- single line comment` `{- multi-line, {- nested, -} comment -}`
Function definitions	

`f p1`
`  # q1 = e1`
`  = e`
	`f p1`
	`  = e[x := x']`
	`  where q1[x := x'] = e1 -- for each x ∈ var(q1) ∩ var(e1)`